Université Lille1 Sciences et Technologies

UFR IEEA

AAC

Durée 1h 30

4 novembre 2013
Master d'Informatique-S1
Elémnts de correction du DS

Les algorithmes seront écrits en pseudo-langage ou dans un langage de votre choix (C, Java, Caml). La clarté de vos réponses et de votre code sera prise en compte.

Exercice 1 : Le B.A.-Ba.

▷ **Question 1:** Pour chacune des affirmations suivantes, dire si elle est vraie ou fausse (il n'est pas demandé de justifier, mais les réponses fausses pourront être comptées "négativement"):

$n^2 \in O(n^3)$? V
$n^3 \in O(n^2)$? F
$n \log n \in O(n^2)$? V
$n + 2\log n \in O(n)$? V
$n + \sqrt{n} \in \Theta(n)$? V
$n\sqrt{n} \in \Theta(n)$? F

▷ **Question 2:** Pour chacun des deux algorithmes suivants, dire si sa complexité (temporelle) dans le pire des cas est en $\Theta(log\ n)$, $\Theta(n)$, $\Theta(n\ log\ n)$ ou $\Theta(n^2)$, en justifiant brièvement:

```
int T1(int n){
  int c=0;
  for (int i=1;i<n;i=i*2)
    for (int j=n;j>0;j=j-10) c++;
  return c;}
```

$\Theta(n\ log\ n)$

```
int T2(int n){
  int c=0;
  for (int i=n;i>0;i--)
    for (int j=i; j<2*i;j++) c++;
  return c;}
```

$\Theta(n^2)$

▷ **Question 3:** Pour les deux algorithmes récursifs suivants, estimer l'ordre de grandeur de la complexité temporelle en évaluant l'ordre de grandeur du nombre d'appels récursifs effectués:

```
int A1(int n){
  if (n>2) return 1+A1(n-3);
  else return 0; }
```

$\Theta(n)$ avec environ $n/3$ appels récursifs

```
int A2(int n){
  if (n>1) return 1+A2(n-2)*A2(n-2));
  else return 0;}
```

$\Theta(2^{n/2})$.

L'arbre des appels est un arbre binaire complet d'hauteur environ $n/2$.

L'équation de récurrence pour $B(n)$, le nombre d'appels à A lors de l'exécution de $A(n)$ est:

$B(n) = 2 * B(n-2) + 1$ si $n > 1$, $B(n) = 0$ si $n \leq 1$.

On obtient $B(n) = 2^{n/2+1)} - 1$.

Exercice 2 : Stratégie optimale

Considérons -encore- une rangée de n pièces de valeur $v_1, \ldots, v_n$ et le jeu suivant à 2 joueurs; à chaque tour, le joueur peut prendre soit la pièce la plus à gauche, soit la pièce la plus à droite. Le jeu s'arrête quand il n'y a plus de pièce. L'objectif est bien sûr de maximiser la somme totale des pièces ramassées.

On cherche à calculer le gain optimal du joueur A, **en considérant que la stratégie du joueur B est la stratégie gloutonne, c'est à dire qu'il choisit toujours de prendre la pièce de plus grande valeur**.

Par exemple, pour $8, 15, 3$, si c'est le tour du joueur B il prendra 8.

▷ **Question 1:** Montrer que pour A, la stratégie gloutonne consistant à prendre parmi les deux pièces possibles, celle à plus forte valeur, n'est pas optimale si il joue face à B.

Attention: optimale ne veut pas nécessairement dire gagnante; elle veut dire maximisant le gain. Contre-exemple: 3, 8, 16, 4: si A commence et prend la stratégie gloutonne: A prend 4, B prend 16, A prend 8, B prend 3 et gagne avec 19 contre 12 pour A. si A commence en prenant 3, B prend 8, A prend 16, B prend 4 et A gagne avec 19 contre 12 pour B.

▷ **Question 2:** Proposer un algorithme qui calcule le gain optimal pour A face à B. La donnée est donc le nombre de pièces et le tableau de leurs valeurs.

Attention: Comme B applique la stratégie gloutonne, on ne pouvait a priori appliquer exactement ce qui avait été vu en TD puisqu'alors on supposait que A et B appliquaient la même stratégie optimale.

Exemple, si on a 5 pièces: 3, 1, 10, 4, 2.

Si B est glouton, A peut gagner en prenant d'abord la pièce 2.

Si B est optimal, A ne peut gagner , mais pour maximiser son gain, il doit prendre d'abord 3.

On a donc: La version récursive "naïve":

```
int  gain(int g, int d) {
   if (g>= d-1) return Math.max(Pieces[g],Pieces[d]);
   int g1= Pieces[g];
   if (Pieces[g+1]>Pieces[d]) g1+=gain(g+2,d); else g1+=gain(g+1,d-1);
   int g2= Pieces[d];
   if (Pieces[g]>Pieces[d-1]) g2+=gain(g+1,d-1); else g2+=gain(g,d-2);
   return Math.max(g1,g2);
}
```

Elle est exponentielle; la version dynamique est en $\Theta(n^2)$:

```
int Calcul(){
   int Gain[][]=new int[nb][nb];
   for (int g=0; g<nb; g++)
       Gain[g][g]=Pieces[g];
   for (int g=0; g<nb-1; g++)
       Gain[g][g+1]=Math.max(Pieces[g],Pieces[g+1]);
   for (int n=2; n<nb; n++)
       for (int g=0; g<nb-n; g++) {
           int d=g+n;
           int g1= Pieces[g];
           if (Pieces[g+1]>Pieces[d]) g1+=Gain[g+2][d]; else g1+=Gain[g+1][d-1];
           int g2= Pieces[d];
           if (Pieces[g]>Pieces[d-1]) g2+=Gain[g+1][d-1]; else g2+=Gain[g][d-2];
           Gain[g][d]=Math.max(g1,g2);
       }
   return Gain[0][nb-1];
}
```

<u>Exercice 3</u> : Chocolate Addiction

Vos chargés de TD ont développé une légère addiction pour le chocolat. Ils se sont fait livrer une tablette de chocolat *infiniment grande*!!! Pour s'y repérer, on désigne chaque carré par ses coordonnées (i, j). Le carré en haut à gauche est le carré $(0, 0)$. La tablette s'étend à l'infini vers le bas et vers la droite. La première coordonnée grandit vers la droite, et la deuxième grandit vers le bas. Pour des raisons pratiques on ne peut accéder à la tablette qu'au travers d'une API qui comporte deux fonctions :

1. `Manger(i,j)` : les chargés de TD appellent cette fonction pour manger d'un seul coup l'ensemble les carrés encore présents dans la tablette qui sont au dessus et à gauche de (i, j).
2. `Tester(i,j)` : la responsable du cours appelle cette fonction pour tester si le carré de coordonnées (i, j) est encore présent dans la tablette ou pas.

Pour fixer les idées, dans la figure A ci-dessous, on distingue le bord de la tablette (le chocolat se trouve en bas et à droite de la frontière, donc). `Tester(3,7)` doit renvoyer "faux", car le carré $(3, 7)$ ne fait plus partie de la tablette. Par contre `Tester(10,10)` doit renvoyer "vrai". On obtient la figure B en exécutant `Manger(8,5)` sur la tablette de la figure A. Le chargé de TD mange tous les 12 carrés grisés d'un seul coup !

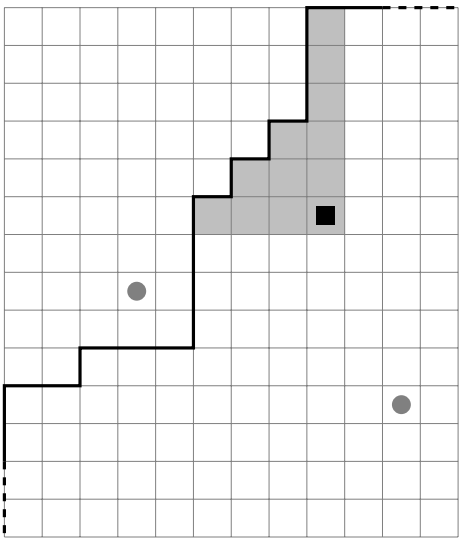

Figure A

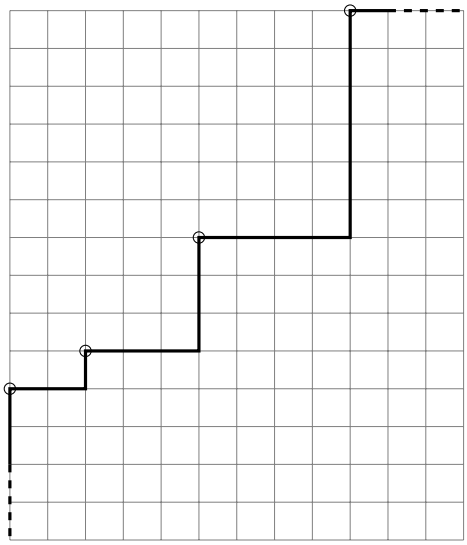

Figure B

L'objectif de cet exercice est de donner des algorithmes qui permettent d'exécuter les deux fonctions de cette API. Pour vous aider, voici une remarque importante : il suffit de stocker les coordonnées des "coins" de la tablette, c'est-à-dire les sommets qui sont entourés dans la figure B. Un "coin" est un carré qui appartient à la frontière (c.a.d. il n'y a plus de chocolat à gauche ou il n'y en a plus au dessus). On exige qu'un coin ne soit pas à droite et en dessous d'un autre coin. Par exemple, dans la figure B, l'angle qui se trouve à $(9, 6)$ n'est pas un coin, car il est "couvert" par le coin $(5, 6)$. Lorsque la tablette est livrée, il n'y a qu'un seul coin en $(0, 0)$.

Préambule

▷ **Question 1:** Est-ce que l'opération `Manger(i,j)` peut faire diminuer le nombre de coins ? De combien de coins au maximum? Est-ce qu'elle peut le faire augmenter ? De combien de coins au maximum? Illustrez vos réponses par des exemples.

`Manger(i,j)` peut faire diminuer le nombre de coins: il peut faire disparaître tous les coins existants, en créant deux nouveaux. Donc, il peut faire diminuer de nbcoins-2 au plus. Il peut faire augmenter le nombre de coins d'au plus 1.

▷ **Question 2:** Y a-t-il toujours un coin dont la première coordonnée est nulle ? Et la deuxième ? oui pour les deux, la tablette étant infinie.

Plus précisément, on peut montrer que la propriété : "Il y a un et un seul coin de première coordonnée nulle" reste toujours vraie.

Au départ: elle est vraie avec le coin (0,0).

Induction: Supposons qu'elle soit vraie. Soit donc $(0, b)$ ce coin. Exécutons $Manger(i, j)$: si $j < b$ le coin $(0, b)$ reste. Sinon, il est détruit et on crée le nouveau coin $(0, j + 1)$ (et pas d'autre coin de première coordonnée nulle).

Par symétrie, la propriété " Il y a un et un seul coin de deuxième coordonnée nulle"" reste toujours vraie.

Tester et Manger

On suppose que les coins sont stockés dans un tableau, et que ce tableau est trié selon la première coordonnée des coins. Un coin sera représenté par la structure de votre choix. Par exemple, pour la figure B on aurait :

```
tab = { (0,10), (2,9), (5,6), (9,0) }
```

▷ **Question 3:** Donnez un algorithme pour `Tester(i,j)` qui s'exécute en temps linéaire en le nombre de "coins".

Plusieurs versions possibles, par exemple:

```
boolean  tester(int i, int j){
     for (int nc=0; nc<nbCoins;nc++)
         if ((Coins[nc].x<=i) && (Coins[nc].y<=j)) return true;
     return false;
 }

boolean testerbis(int i, int j){
```

```
     for (int nc=0; nc<nbCoins;nc++)
            if (Coins[nc].y<=j) return(Coins[nc].x<=i);
    throw new IllegalArgumentException("INVALID"); //Coins[nbCoins-1].y=0<=j
}
```

▷ **Question 4:** Expliquez pourquoi dans le tableau, les deuxièmes coordonnées se trouvent triées par ordre décroissant.

si on a un coin (i,j), on ne peut pas avoir un coin différent (i',j') avec i'<=i et j'<=j. Donc si i' <=i, j'>j q.e.d.

▷ **Question 5:** Donnez un algorithme pour `Tester(i,j)` qui s'exécute en temps *logarithmique* en le nombre de "coins". Avec une recherche dichotomique par exemple du premier coin du tableau tel que (Coins[nc].y<=j):

```
boolean testerdicho(int i, int j){
    int g=0;
    int d=nbCoins-1;
    while (g < d) { // inv: g<=d, avant g Coins[nc].y >j, Coins[d].y <=j
        int mid=(g+d)/2;
        if (Coins[mid].y>j) g=mid+1; else d=mid;
    }
    return(Coins[g].x<=i);
}
```

▷ **Question 6:** Donnez un algorithme pour `Manger(i,j)`, et donnez sa complexité.

Le schéma est par exemple:
//suppose que tester(i,j) retourne Vrai
k = 0, Tant que (coin[k].j >y+1), incrémenter k;
l = k+1, tant que (l<n et coin[l].i < =x+1) incrémenter l;
// les coins [0; k[sont conservés, les coins [k; l[sont détruits, les coins [l; n[sont conservés
Initialiser une nouvelle liste L de coins, initialement vide
Pour m de 0 à k-1, copier coin[m] dans L
// fabrique les nouveaux coins
ajouter (coin[k].i, y+1) dans L //peut être le même qu'avant si y+1=coin[k].j
ajouter (x, coin[l-1].j) dans L //peut être le même qu'avant si x+1=coin[l-1].i
Pour m de l à n-1, copier coin[m] dans L
renvoyer L

par exemple en pseudo-Java:

```
//précondition: tester(i,j) est vrai
Coin[] manger(int i, int j){
    int c=0;
    int nouvc=0;
    Coin[] nouv= new Coin[nbCoins+1]; //au maximum 1 coin créé
    while (coins[c].y>j+1) nouv[nouvc++]=coins[c++]; //coin inchangé
    // c le plus petit tel que  coins[c].y<=j+1,
    nouv[nouvc++]= new Coin(coins[c].x,j+1);
    //nouveau coin, peut être égal à coins[c], si coins[c].x=j+1.
    while ((c<nbCoins) && (coins[c].x<=i+1))  c++; //coin détruit
   // c le plus petit tel que  coins[c].x>i+1
    nouv[nouvc++]= new Coin(i,coins[c-1].y);
  //nouveau coin, peut être égal au coin détruit coins[c-1], si coins[c-1].x=i+1
    while (c<nbCoins)  nouv[nouvc++]=coins[c++]; //coin inchangé
    return Arrays.copyOfRange(nouv,0,nouvc);
  }
```

La complexité est linéaire en le nombre de coins

▷ **Question 7:** [Bonus] Quelle structure de données pour les coins pourrait améliorer la complexité de `Manger(i,j)`?

Par exemple, un arbre type AVL, mais cela impose d'utiliser des primirves assez compliquées sur les AVL...

Lors d'une opération Manger(), les coins qui disparaissent sont consécutifs. Pour faire à la fois tester() et manger() en temps logarithmique, il faut une structure de donnée qui permette à la fois : A) d'identifier le premier coin mangé, ainsi que le dernier, en temps O(log n) B) de supprimer tous les coins dans l'intervalle en temps O(log n) C) de rajouter les deux nouveaux coins fabriqués en temps O(log n)

Là-dedans, seul le point B est vraiment problématique. En effet, avec des arbres binaires de recherche équilibrés (genre AVL), on peut faire A et C avec des recherches, des insertions et des deletions en temps O(log n). On peut aussi faire B même si ce n'est pas facile. En fait, pour pouvoir faire B, il faut avoir à sa disposition, en plus des fonctions habituelles, deux nouvelles opérations sur les AVLs :

$Split(T, x)$ qui renvoie deux AVL contenant respectivement les noeuds de T inférieurs ou égaux à x d'un côté, et supérieurs à x de l'autre

$Join(T_1, T_2)$ qui renvoie l'AVL contenant les noeuds de T_1 et T_2, sachant que tous les noeuds de T_1 sont plus petits que ceux de T_2.

Avec ces deux opérations (2 Split et 1 Join), on peut éliminer d'un seul coup tous les coins mangés. Ces deux opérations sont un peu compliquées, mais on peut les accomplir en temps O(log n). Elles sont décrites dans la partie consacrée aux AVL de la bible "The Art of Computer Programming" de Don Knuth, volume 3.